NB results for lapwing data

Philip Dixon

9/5/2021

Interpreting results from glm(), glm.nb(), optim() and optimize()

using 1960's Northern lapwing data That is the c2 vector below.

```r
# No. lapwing (Vanellus vanellus) on Skokholm, 1930's vs 1960's
c1 <- c(10, 11, 12, 10, 8, 6, 5, 3, 5, 4)
c2 <- c(25, 17, 20, 4, 7, 18, 27, 18, 18, 10)
```

Using glm() to fit a Poisson model

Essentially like lm() to fit a linear model (normally distributed errors), with a few extra things to take care of.

- Need to add family= to specify the error distribution. The default is gaussian = normal, which you almost certainly don't want. You would just use lm() to fit data with normally distributed errors.
- You need to be aware of the link function. The model models the mean response (for a group in an ANOVA or for a combination of X variables in a regression), but it does it indirectly through a link function. The default link depends on the distribution. For a normal distribution, the default link is identity, so the model is
$$\mu_i = \beta_0 + \beta_1 X_i.$$
For a Poisson or Negative Binomial distribution, the default link is the log, so the model is
$$\log \mu_i = \beta_0 + \beta_1 X_i.$$

If you want to estimate means, you need to backtransform results from the fitted glm, because of the link function. This is true even when the model has only an intercept.

Here's how to estimate the mean for the 1960's lapwings:

```r
P1960.fit <- glm(c2 ~ 1, family=poisson)
```

If the response variable and possibly covariates are in a data frame, you would add data= to the call (just like lm()).

You have all the usual helper functions, just like with lm():

- printing the glm result: You get basic output
- coef(): gives you just the estimates
- summary(): You get more information about each coefficient and the overall model
- anova(): You get sequential tests of hypotheses
- logLik(): gives you the log likelihood. Note the 2nd Capital L
- AIC(): gives you the AIC statistic
- resid(): gives you the residuals. There multiple ways to define residuals for a glm() model. The default is "deviance", which is generally a good choice. We'll see something better (PIT residuals) soon, but that's not implemented in residuals.glm()

- predict(): gives you predicted values. Default is type='link' to give you "linear predictors", i.e., on the log scale for a Poisson or NB distribution. type='response' gives you predictions on the data scale, i.e. counts.

These are demonstrated here:

```
P1960.fit
```

```
##
## Call:  glm(formula = c2 ~ 1, family = poisson)
##
## Coefficients:
## (Intercept)
##       2.797
##
## Degrees of Freedom: 9 Total (i.e. Null);  9 Residual
## Null Deviance:       34.11
## Residual Deviance: 34.11     AIC: 81.28
```

```
coef(P1960.fit)
```

```
## (Intercept)
##    2.797281
```

```
summary(P1960.fit)
```

```
##
## Call:
## glm(formula = c2 ~ 1, family = poisson)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.6759  -1.2417   0.3889   0.7415   2.3921
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  2.79728    0.07809   35.82   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 34.114  on 9  degrees of freedom
## Residual deviance: 34.114  on 9  degrees of freedom
## AIC: 81.283
##
## Number of Fisher Scoring iterations: 4
```

```
anova(P1960.fit)
```

```
## Analysis of Deviance Table
##
## Model: poisson, link: log
##
## Response: c2
##
## Terms added sequentially (first to last)
##
```

```
##
##      Df Deviance Resid. Df Resid. Dev
## NULL                     9     34.114
```

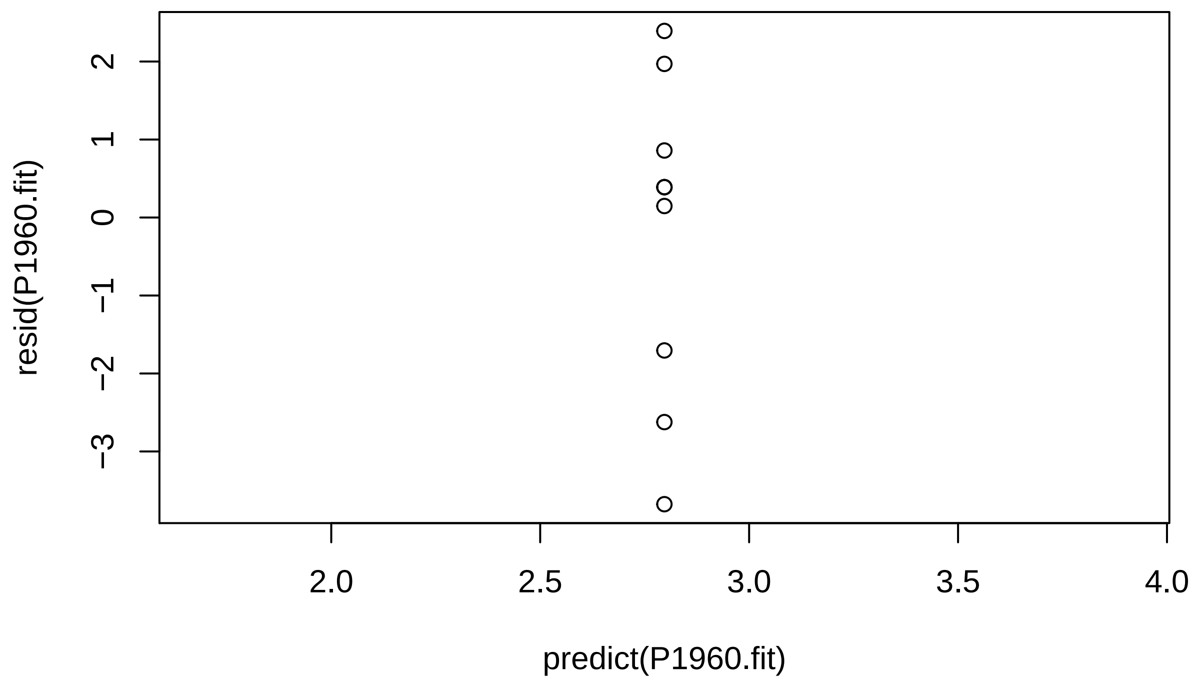
```
logLik(P1960.fit)
```

```
## 'log Lik.' -39.64166 (df=1)
```

```
AIC(P1960.fit)
```

```
## [1] 81.28332
```

```
plot(predict(P1960.fit), resid(P1960.fit))
```



If the model fits, the deviance residuals are approximately standard normal, i.e. mean=0, sd = 1. That means approximately 95% of the values should be between -2 and 2. This approximation is best when the counts are large. Here, it looks like the residuals, and hence the data, are more variable than they should be for a Poisson distribution. Note that 3 of the 10 residuals exceed -2 or +2 and one is less than -3.

You can provide any linear model on the right-hand side of the model equation. This means you can fit two-sample comparisons, ANOVA models, simple or mulitple regression models, and combinations. Just make sure that if you want to compare groups, the X variable (right-hand side) is a factor variable.

To demonstrate, using the comparison between 1960's and 1930's abundance:

```
bothperiods <- data.frame(period = rep(c(1930, 1960), c(10,10)), count=c(c1, c2))
bothperiods[c(1:2, 11:12),]
```

```
##   period count
## 1   1930    10
## 2    1930    11
## 11   1960    25
## 12   1960    17
```

```
bothperiods$period.f <- factor(bothperiods$period)
#  needed because period was a continuous variable, so count ~ period would be a regression
both.glm <- glm(count ~ period.f, family=poisson, data=bothperiods)
summary(both.glm)
```

```
##
## Call:
## glm(formula = count ~ period.f, family = poisson, data = bothperiods)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.6759  -1.0460   0.3033   0.9067   2.3921
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   2.0015     0.1162  17.217  < 2e-16 ***
## period.f1960  0.7958     0.1400   5.683 1.33e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 81.928  on 19  degrees of freedom
## Residual deviance: 47.033  on 18  degrees of freedom
## AIC: 133.92
##
## Number of Fisher Scoring iterations: 4
```

```
# to see how R constructs a covariate from the factor variable:
cbind(period=bothperiods$period, model.matrix(both.glm))[c(1:2, 11:12),]
```

```
##    period (Intercept) period.f1960
## 1    1930           1            0
## 2    1930           1            0
## 11   1960           1            1
## 12   1960           1            1
```

You see that period.f is defined as I did in lecture: 0 for 1930's, 1 for 1960's observations. Hence, the coefficient for period.f is log(1960 mean / 1930 mean) and the intercept coefficient is log 1930 mean. Exponentiating them gives the 1930's mean and the ratio (1960's / 1930's).

This is the default contr.treatment definition of a factor variable. There are others; all work in glm(), but using others assumes you know what you want because you are familiar with some linear models theory. The functions in the emmeans library, illustrated by code in lapwing.r, allow you to get what you might want without knowing the linear models theory (but knowing some things about emmeans).

```
exp(coef(both.glm))
```

```
##  (Intercept) period.f1960
##     7.400000     2.216216
```

The other useful stuff in the summary results are:

- In the coefficients table:

- the estimated coefficients, on the log scale
- standard errors of the log scale coefficients
- z statistics testing coefficient = 0 = estimate / se
- P-value for that test, assuming the asymptotic normal distribution for the log scale estimate
- Elsewhere:
- Null deviance: -2 lnl for a model with only the intercept
- Residual deviance: -2 lnl for the specified model
- AIC value for the fitted model

You could construct a likelihood ratio test using the change in deviance to test the null hypothesis that all coefficients other than the intercept = 0. Most interesting when the model is a single effect, i.e. one covariate or one grouping variable. You have to do the subtraction and comparison to a Chi-square distribution "by hand".

Using glm.nb() to fit a negative binomial model

The glm() function fits many different distributions, but it requires that they be in a class of statistical distributions called the exponential family. The negative binomial distribution is in the exponential family when the over dispersion coefficient is specified, but not in general. Hence, fitting a negative binomial model requires a new function. I use glm.nb() in the MASS library. This uses a log link by default. Unlike glm(), you do not need to specify the family because glm.nb() only fits a negative binomial. There are almost certainly other functions in other packages, but I haven't tried to find or understand them.

```
# the glm.nb() is in the MASS library
#  included with the standard distribution of R, so don't need to include.packages() it.
library(MASS)

# fitting the log mean for 1960
NB1960 <- glm.nb(c2 ~ 1)

# fitting both groups
NBboth <- glm.nb(count ~ period.f, data=bothperiods)
summary(NBboth)
```

```
##
## Call:
## glm.nb(formula = count ~ period.f, data = bothperiods, init.theta = 8.418891076,
##     link = log)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4309  -0.7888   0.1910   0.6453   1.3150
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.0015     0.1593  12.560  < 2e-16 ***
## period.f1960   0.7958     0.2082   3.821 0.000133 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Negative Binomial(8.4189) family taken to be 1)
##
##     Null deviance: 34.935  on 19  degrees of freedom
## Residual deviance: 20.138  on 18  degrees of freedom
## AIC: 125.7
```

```
##
## Number of Fisher Scoring iterations: 1
##
##
##               Theta:  8.42
##           Std. Err.:  4.71
##
##  2 x log-likelihood:  -119.70
```

The output is very similar to that from glm(). The important new thing is the estimate of Theta, θ. That is 8.42 here. That is what I have called r, i.e. the overdispersion parameter where the variance of Y $= \mu + \mu^2/\theta$). So very large, ideally ∞, represents a Poisson distribution.

glm.nb() assumes a common overdispersion for all observations. That is, for both groups in this problem.

If you compare the glm.nb() results to the earlier results from glm(), you'll see the same estimates but larger standard errors from glm.nb(). That represents the additional uncertainty modeled by the negative binomial distribution.

Using optim and optimize to fit a NB distribution to the lapwing data

This does not require that a distribution be in the exponential family, so code such as that given below can be used (with appropriate changes) for any distribution.

The log likelihood function is defined in lnlNB.r. This code is not duplicated in the output to save space.

Arguments to the log likelihood (lnL) function are:

- 1st argument: parameter values at which to calculate lnL
- 2nd argument: data to use in calculating lnL
- 3rd or later: any additional information used to calculate lnL

Only the first argument is required. Some folks include the data in the lnL function. My preference is to pass in the data as a named value. The variable names (param, y) are arbitrary. But, if you rename y, you need to change aspects of the call to optim()

The trace= argument is something I use to illustrate numerical optimization. For my functions, trace=T prints out the values and lnL for each use of the lnL function. This is totally optional and can be omitted without consequence.

The lnL function should return a scalar value for that set of parameter values

For example, to calculate the log likelihood for a NegBin distribution with mu=10, r=3 fit to the 1960's data:

```
lnlNB(c(10, 3), y=c2)
```

```
## [1] -1190.873
```

By default, trace is FALSE.

Using optim()

optim() requires 2 arguments:

- 1st argument: vector of starting values for the parameters, length = # parameters
- 2nd argument: the name of the function to find the minimum or maximum of

I customarily add additional arguments:

- y=: provides the data to be used by lnlNB0() or lnlNB(). Name (y) must match the name of the argument of lnlNB0() or lnlNB(). If you change the name in the definition of lnlNB0() or lnlNB(), e.g. to lnlNB <- function (. . . , data=, . . .), you must use data= in the call to optim, so optim can pass along the data.

- method='BFGS': tells optim() to use the BFGS algorithm. This is my "go to" numerical maximizer. Generally works quickly and robustly. The default is Nelder-Mead.
- control=list(fnscale=-1): tells optim() to maximize the lnL. optim() is hard-wired to minimize the function. We want to maximize the function. Control= specifies a list of options you can change. fnscale= tells optim to multiple the result from lnlNB() by -1. Minimizing the negative log likelihood = maximimizing the log likelihood. An alternative is to write a function that returns the negative log likelihood and not use fnscale=.
- hessian=T: tells optim() to include the estimated hessian matrix. This has the 2nd derivatives of lnL, evaluated at the mles. This will be used to estimate the standard errors of the estimates. Can also use to get the correlations between estimates.

This code tells optim to use the lnlNB() function, starting at mu=5, r=20, to fit the c2 data (1960's) using the BFGS method, and include the hessian matrix in the result.

Why are the parameters mu=5, r=20 and not r=5, mu=20? Because of how I wrote lnlNB. The first parameter value is mu; the second is r. You choose - just remember the order. This is one reason why the very first part of my log likelihood functions extracts and gives interpretable names to the components of the parameter vector.

Why are the starting values log(5) and log(20)? Again, because of how I wrote lnlNB. The lnL is defined for parameters on the log scale.

```
fit1960 <- optim(c(log(5), log(20)), lnlNB, y = c2, method='BFGS',
  control=list(fnscale=-1), hessian=T)

fit1960
```

```
## $par
## [1] 2.797387 1.809546
##
## $value
## [1] -34.24096
##
## $counts
## function gradient
##       22       11
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##              [,1]        [,2]
## [1,] -44.50081882 -0.00344196
## [2,]  -0.00344196 -2.41068257
```

The results are:

- par: the estimated parameter values at the maximum
- value: the value of lnL at the maximum
- counts: information about how hard the function was to fit. The number of times optim called the function to calculate the gradient (1st derivative) and called the function for other purposes. Not

statistically useful, but if these are large, you know your likelihood was hard to optimize for the starting values you provided.

- convergence and message: whether or not the optimization converged and if not, why?
- convergence = 0 is what you want. If so, optim() believes it has found the maximum
- hessian: the matrix of 2nd derivatives. All the diagonal elements should be negative. If not, you have found a minimum or a saddlepoint and when you calculate the variance covariance matrix, you'll have one or more negative variances (bad news).

The result is a list. You can extract each component individually by listname$component, e.g., fit1960$par or fit1960$hessian.

Useful things you can do with the optim() output

- Backtransform the parameters. Here, from log(mu), log(r) to mu, r:
- Calculate the variance covariance matrix of the estimates
- Extract the standard errors of the estimates

These are demonstrated by this code:

```
#  when all parameters are backtransformed the same way
exp(fit1960$par)
```

```
## [1] 16.401741  6.107675
```

```
# when each parameter needs to be backtransformed separately, also names the components
c(mu = exp(fit1960$par[1]),  r = exp(fit1960$par[2]) )
```

```
##        mu         r
## 16.401741  6.107675
```

```
# Get the variance-covariance matrix and save it, then print it
vc <- solve(-fit1960$hessian)
vc
```

```
##               [,1]          [,2]
## [1,]  2.247150e-02 -3.208469e-05
## [2,] -3.208469e-05  4.148203e-01
```

```
# get the standard errors of the estimates as sqrt(diagonal values of the vc matrix)
sqrt(diag(vc))
```

```
## [1] 0.1499050 0.6440655
```

Using optimize() to maximize a one parameter function

The algorithms in optim() work best with 2 or more parameters. If there's only one parameter, there are better ways to find the optimum. The optimize() function implements one very good one, golden section search.

This is illustrated with the lnlNBr() function defined in lnlNB.r and not copied here. This function evaluates the log likelihood for a specified value of r with the mu specified separately.
The arguments of this function are:

- r: a scalar value
- y: the data, i.e. the vector of counts
- mu: a fixed value of the mean count. Because this is specified separately, it will not be optimized. Instead, using the lnlNBr() function will find the best value of r (the 1st argument) given the specified value of mu.
- trace: optional, whether to print values each time the lnlNBr() function is called.

optimize() searches a specified interval for the maximum. This is different from optim() that requires a starting value, but then will search everywhere for the maximum / minimum. With optimize(), you specify the lower and upper bounds of the interval to search. I usually use 'as small as reasonable' and 'as large as reasonable' as the bounds. If that creates numerical issues, I shrink the search interval.

using optimize() optimize() has two (or three) required arguments:

- the 1st argument: the name of the function to calculate the log likelihood
- either interval=: a 2 element vector with the lower and upper bounds of the search interval.
- or lower= and upper= giving the lower and upper bounds of the seach interval

The other arguments are optional (except for maximize, which you always want for a log likelihood function): * maximum=T: maximize the function. The default is to minimize it * other arguments required for your log likelihood function * y=: specifies the data. As with optim, y matches the variable name in the lnlNBr function definition * mu=: specifies parameter values to be held constant. Here, this is only one parameter, mu. This can be a vector; in so, this is "unpacked" by your function.

Here's how to use optimize() to find the mle of r for the 1960's data given mu = 16.4:

```
NB1960r.fit <- optimize(lnlNBr, lower=1, upper=100, y=c2, mu=16.4, maximum=T)
NB1960r.fit
```

```
## $maximum
## [1] 6.107994
##
## $objective
## [1] -34.24096
```

The result is a list with two components:

- maximum: the parameter value (here r, because of how lnlNBr was written) that maximizes the log likelihood
- objective: the log likelihood at the maximum value

These match those from the (mu, r) optimization using optim() because I specified mu as the mle.

things that might go wrong using optimize() Here's what happens if you specify too small a search interval:

```
bad <- optimize(lnlNBr, lower=1, upper=5, y=c2, mu=16.4, maximum=T)
bad
```

```
## $maximum
## [1] 4.999938
##
## $objective
## [1] -34.29016
```

The only sign of a problem is that the reported maximum is at the boundary of the interval you specified. That's the best lnL within the interval. Because the search interval is too narrow, it's not the mle.

Here's what happens if you forget to specify maximum=T

```
bad <- optimize(lnlNBr, lower=1, upper=100, y=c2, mu=16.4)
bad
```

```
## $minimum
## [1] 99.99996
##
## $objective
## [1] -38.26991
```

Again, the only sign of a problem is that the reported maximum is at the boundary of the interval you specified. That's the worst lnL within the interval. Only sign of a mistake is that the estimate is at the boundary.